Verifying an Interactive Consistency Circuit: A Case Study in the Reuse of a Verification Technology

Mark Bickford
Mandayam Srivas

Odyssey Research Associates, Inc.
301A Harris B. Dates Drive
Ithaca, NY 14850.

This talk presented the work done at ORA for NASA-LRC in the design and formal verification of a hardware implementation of a scheme for attaining interactive consistency (byzantine agreement) among four microprocessors. The microprocessors used in the design are an updated version of a formally verified 32-bit, instruction-pipelined, RISC processor, MiniCayuga. The 4-processor system, which is designed under the assumption that the clocks of all the processors are synchronized, provides ''software control'' over the interactive consistency operation. Interactive consistency computation is supported as an explicit instruction on each of the microprocessors. An identical user program executing on each of the processors decides when and on what data interactive consistency must be performed.

This exercise also served as a case study to investigate the effectiveness of reusing the technology which had been developed during the MiniCayuga effort for verifying synchronous hardware designs. MiniCayuga was verified using the verification system Clio which was also developed at ORA. To assist in reusing this technology a computer-aided specification and verification tool was developed. This tool specializes Clio to synchronous hardware designs and significantly reduces the tedium involved in verifying such designs. The talk presented the tool and described how it was used to specify and verify the interactive consistency circuit.

Summary

Achievements

1. Formalization of abstract Byzantine agreement algorithm.

2. Use of this algorithm to specify a hardware device.

3. A mechanically checked proof that the device design is correct.

4. The implementation of the device form the low-level design.

Limitations

1. Assumes synchronized behavior of the processes.

Verifying an Interactive Consistency Circuit:

A Case Study in the Reuse of a Verification Technology

Mark Bickford
Mandayam Srivas

Odyssey Research Associates, Inc.
301A Harris B. Dates Drive
Ithaca, NY 14850.

Objectives of the Work

- Design an efficient hardware implementation for a 4-processor architecture
- Use verified MiniCayuga's in the design
- Verify the design
- Reuse MiniCayuga verification technology
  - A method of modeling synchronous hardware designs in the Clio verification system
  - Formalizing a class of properties most commonly encountered in verifying designs
  - A "standard" proof strategy

Clio: A functional Language Based Verification System

- Caliban: A modern functional language
  eg.: higher order functions, data types, lazy, etc.

```
least P x =  x, P x
             least P x+1
```

- Assertion Level: Full FOPC with equality on Caliban terms

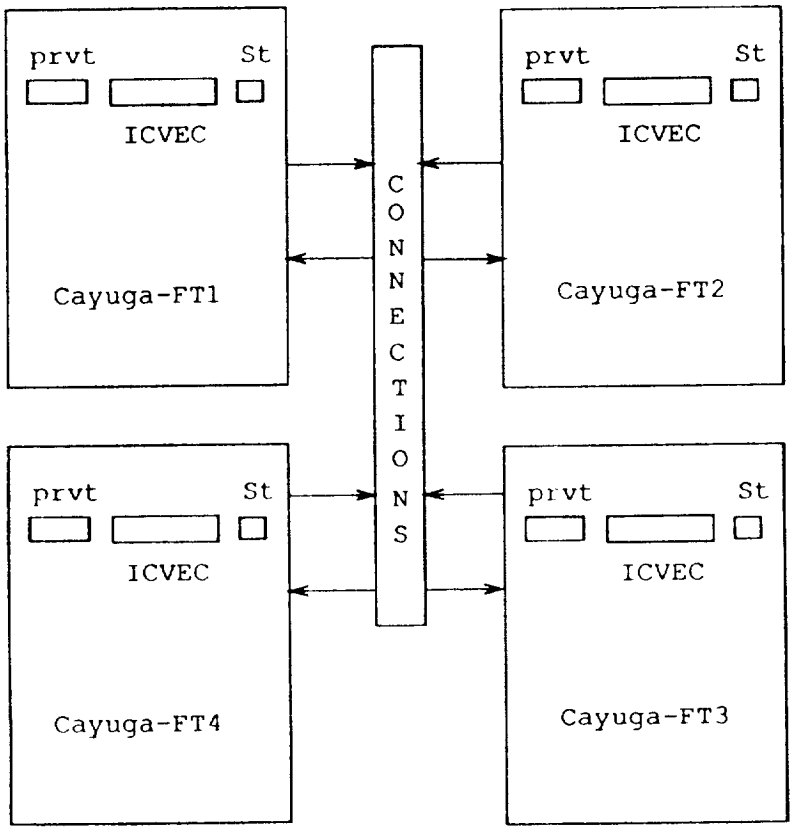
```
Prop := (P)(x) -> [`!(least P x)` = `True`]
                  V `P (least P x)` = `True`
```

- Interactive Theorem Prover
  - rewriting
  - Induction
    - structural
    - Fixed point
  - Other FOPC proof strategies

Presentation Outline

- IC circuit design
- The computer-aided hardware verification tool
- How we verified it
- General observations about the effort

The Hardware Design: Overview

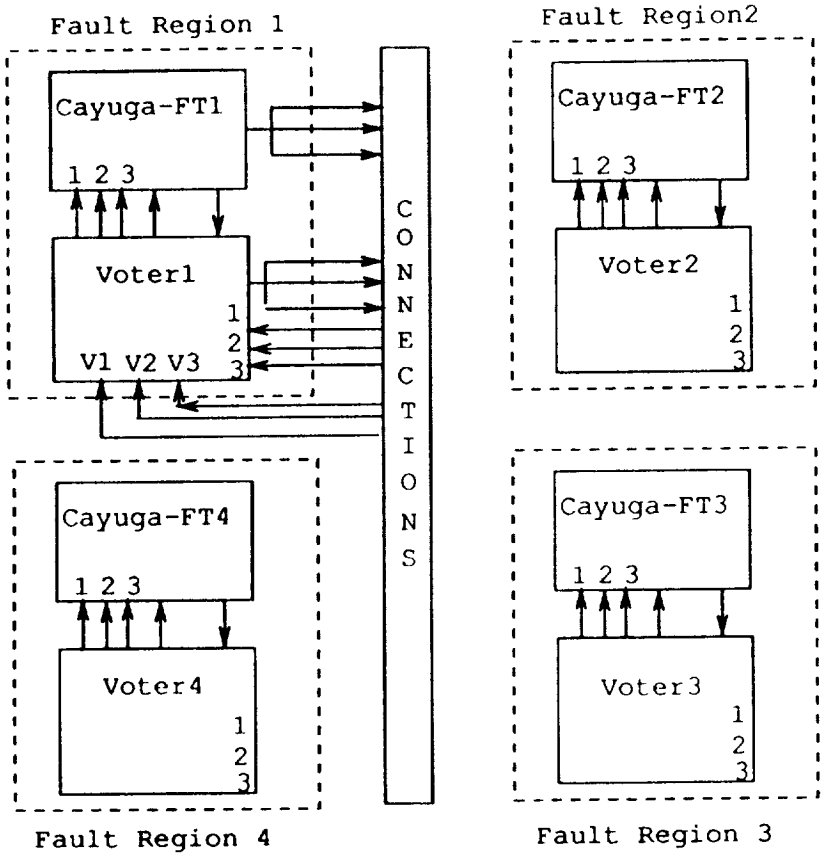


Two new instructions:

```
ICOP REG        - initiates and co-orinates
                  IC computation
MOVE SREG REG - moves special REG to
                  general REG
```

```
|| check if voter is free
Notfree     MOVE STATUS REG1
            JIF REG1 Notfree
            ICOP REG2
|| check if IC computation is complete
Notready    MOVE STATUS REG1
            JIF REG1 Notready
|| move the results of IC to general registers
            MOVE SREG0 REG3
            MOVE SREG1 REG4
            MOVE SREG2 REG5
```

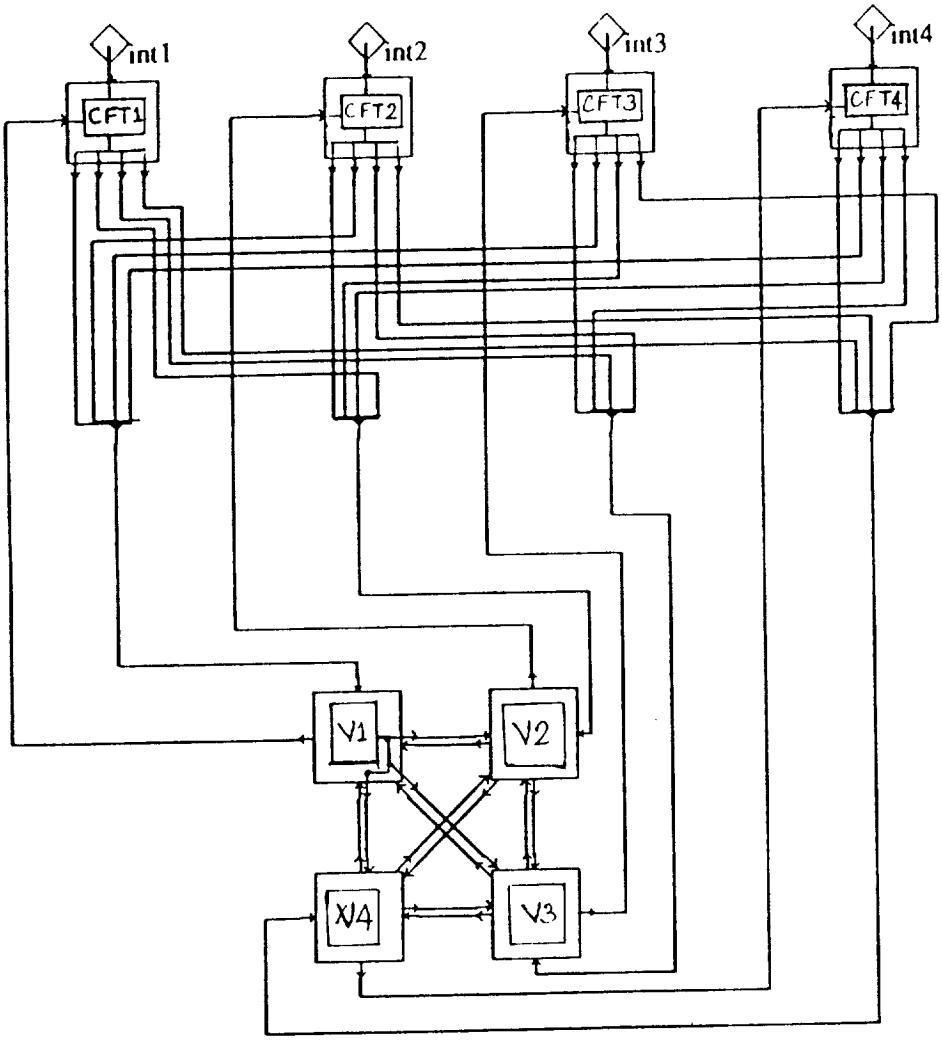
The Hardware Design: Overview



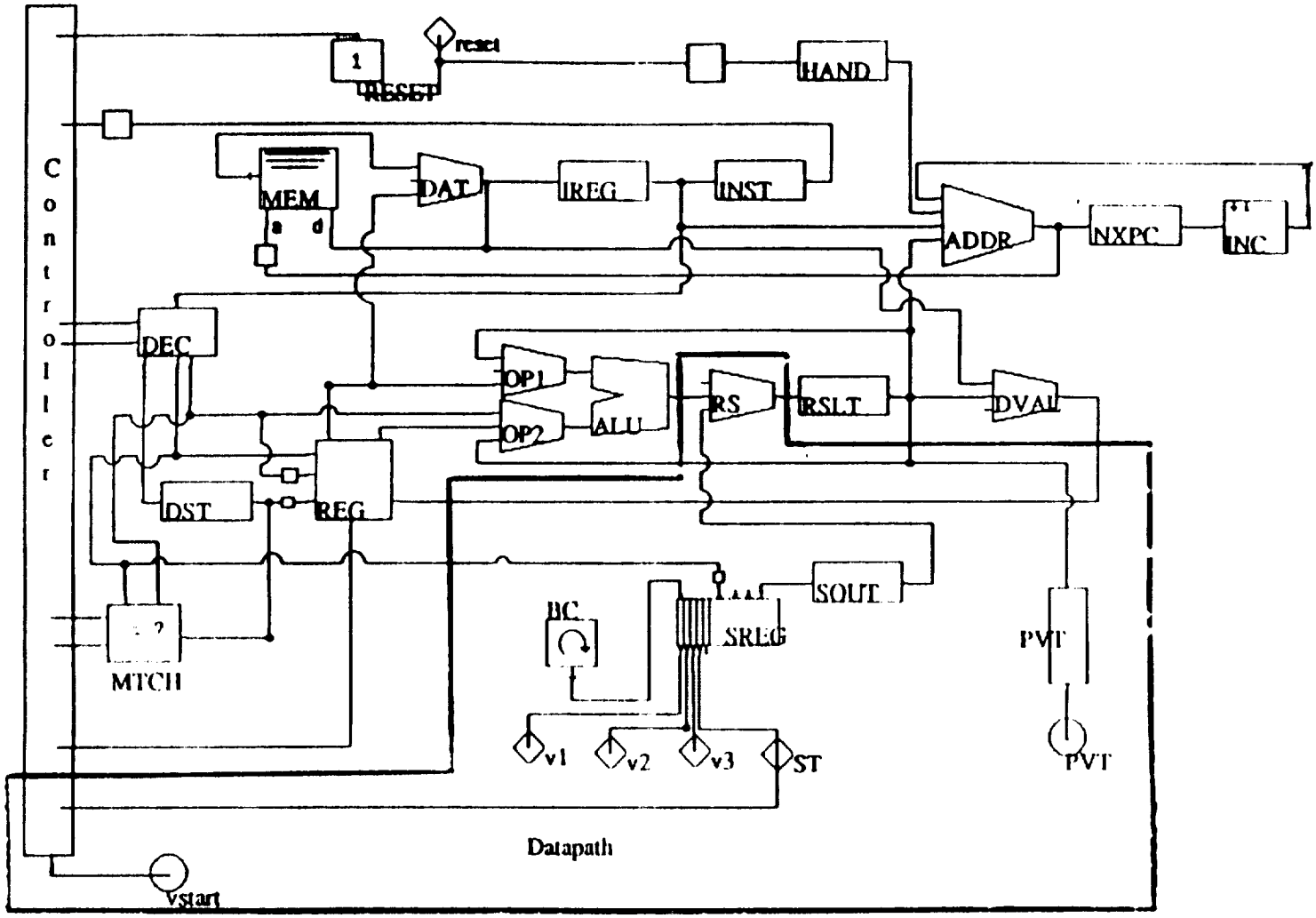
- voter separate from processor: modularity
- point-to-point connection: electrical isolation
- serialize data transfers: number of pins Vs. time
- Fault region: processor, voter, and the connections they feed

- no absolute indexing scheme for processors/voters
  - relative indexing scheme: $succ$, $succ^2$, $succ^3$
  - IC vectors will be stored in the processors in the order of their successors
- Underlying assumption: clocks are synchronized with at most a bounded skew
  - hold sender's signal stable for one phase longer than needed

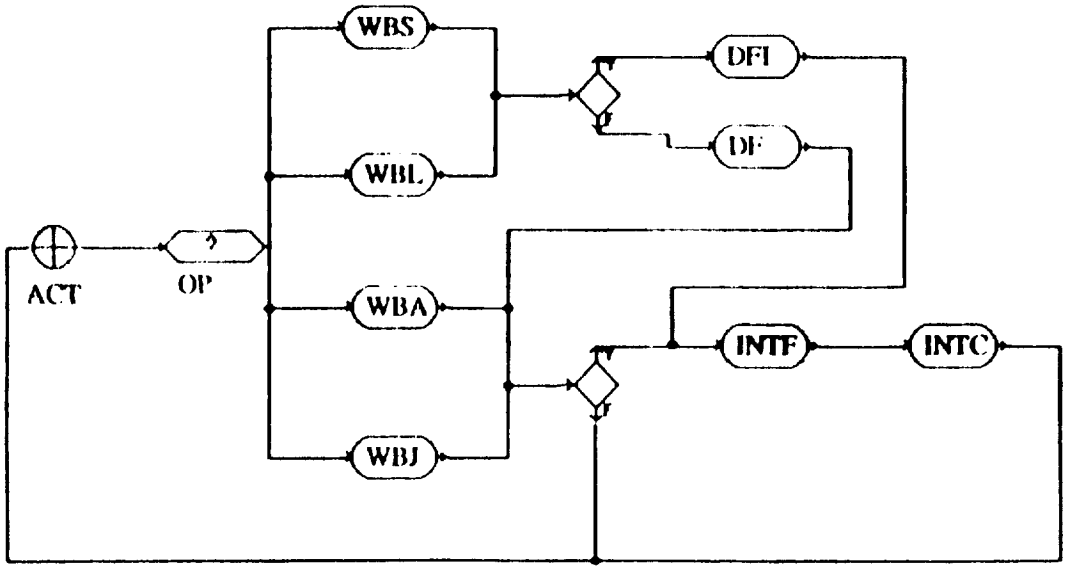
IC System Design Behavior



- *Initiate*: draw the attention of voter (1)
- *Load*: transfer private values (2)
- *Exchange*: exchange received values (6)
- *Compute*: compute and store IC vector (3)



Datapath



Controller State Machine

MiniCayuga Processor: Summary

- Inspired by Cayuga (Cornell University)
- 32-bit RISC processor
- Design characteristics
  - 32 general purpose registers
  - small and simple instruction set
  - 3-stage instruction pipeline: fetch, compute, writeback
  - delayed jump, pipeline stalling, internal forwarding
  - interrupt

What do we prove ?

Assuming

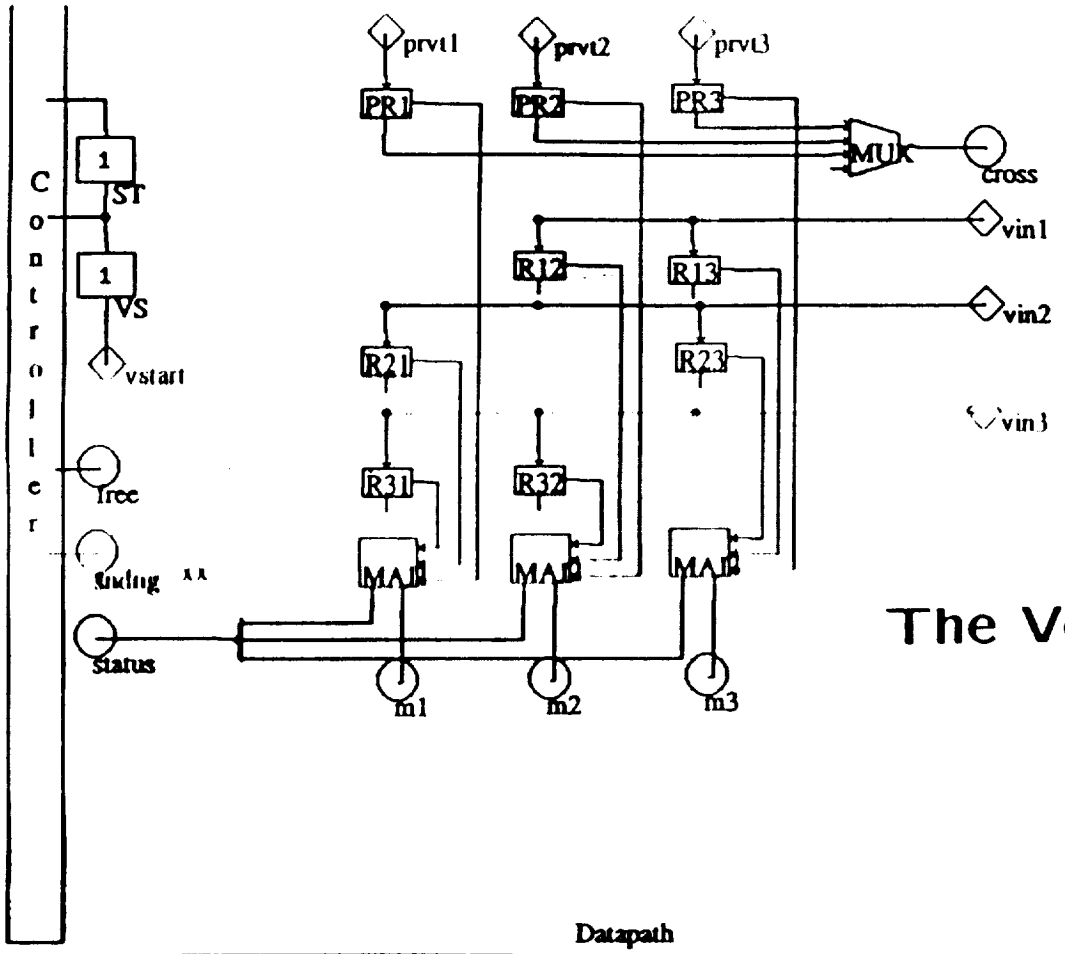
- every Cayuga-FT is about to execute an `ICOP`,
- every Voter is ready to vote, and
- there is at most one faulty region,

then, 12 cycles later the system state will satisfy the following conditions:

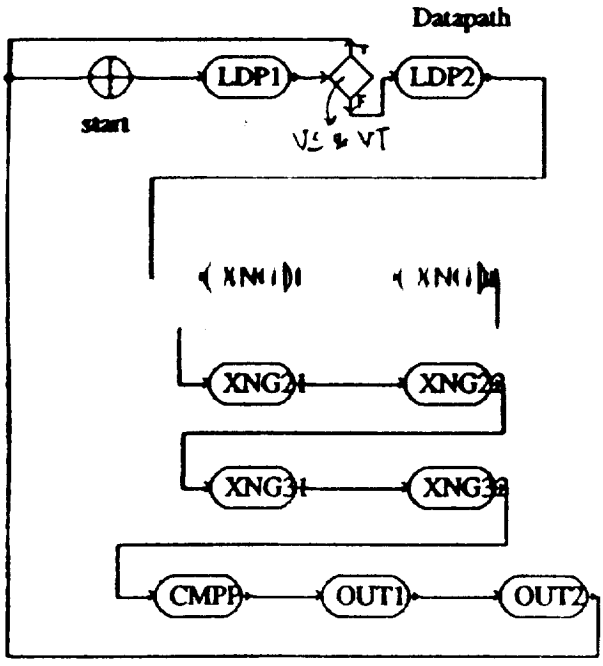
- The IC vectors in the processors are identical "up to rotation."
- The IC vectors are correct w.r.t. to the processor private values 12 cycles earlier.

A Computer-Aided Verification Tool

- Specializes Clio to the domain of *finite state controller systems*
- Design specification generation
- Verification condition formulation
- Automatic proof support



The Voter Circuit



Controller State Machine

Finite State Controller Systems (FSCS)

- Central Controller + Data Path components

- Component behavior is specified as a set of *actions*

- Controller is specified as an FSM which schedules a set of *actions* on the components.

- Timing Model
  - Every transition corresponds to a clock cycle (with multiple phases)
  - An action may have zero or more units (phases) of delay
  - Actions are synchronized with state transitions

Specification technology reused

- a method of formalizing the intended operational model of an FSCS in Caliban/Clio

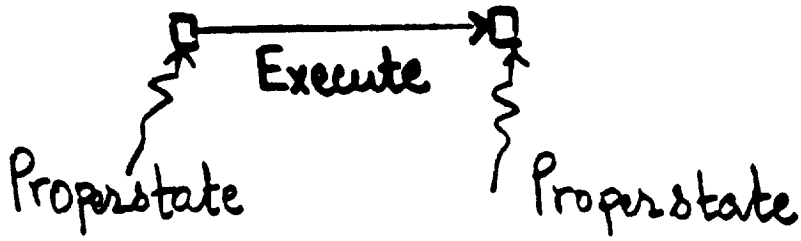
```
designspecgen ::
   data-path-structure ->
      controller-structure ->
         controller-schedule ->
            actions-behavior -> design-spec

Execute :: STATE -> STATE
```

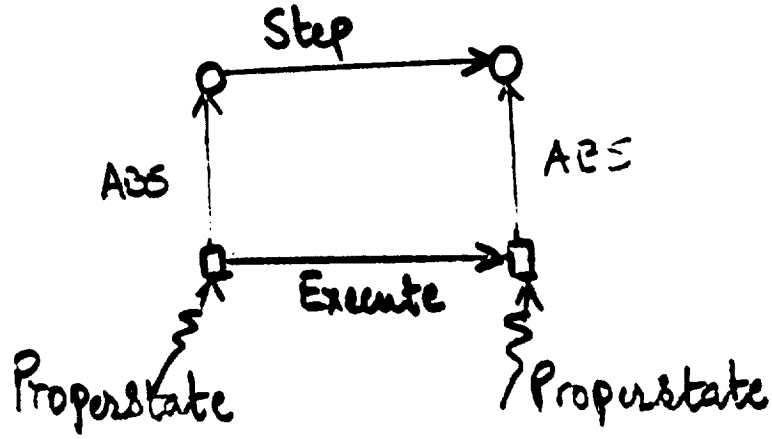
"single clock cycle behavior of design"

Proof technology shared

- Form of the most commonly proved conditions
  - Invariant conditions

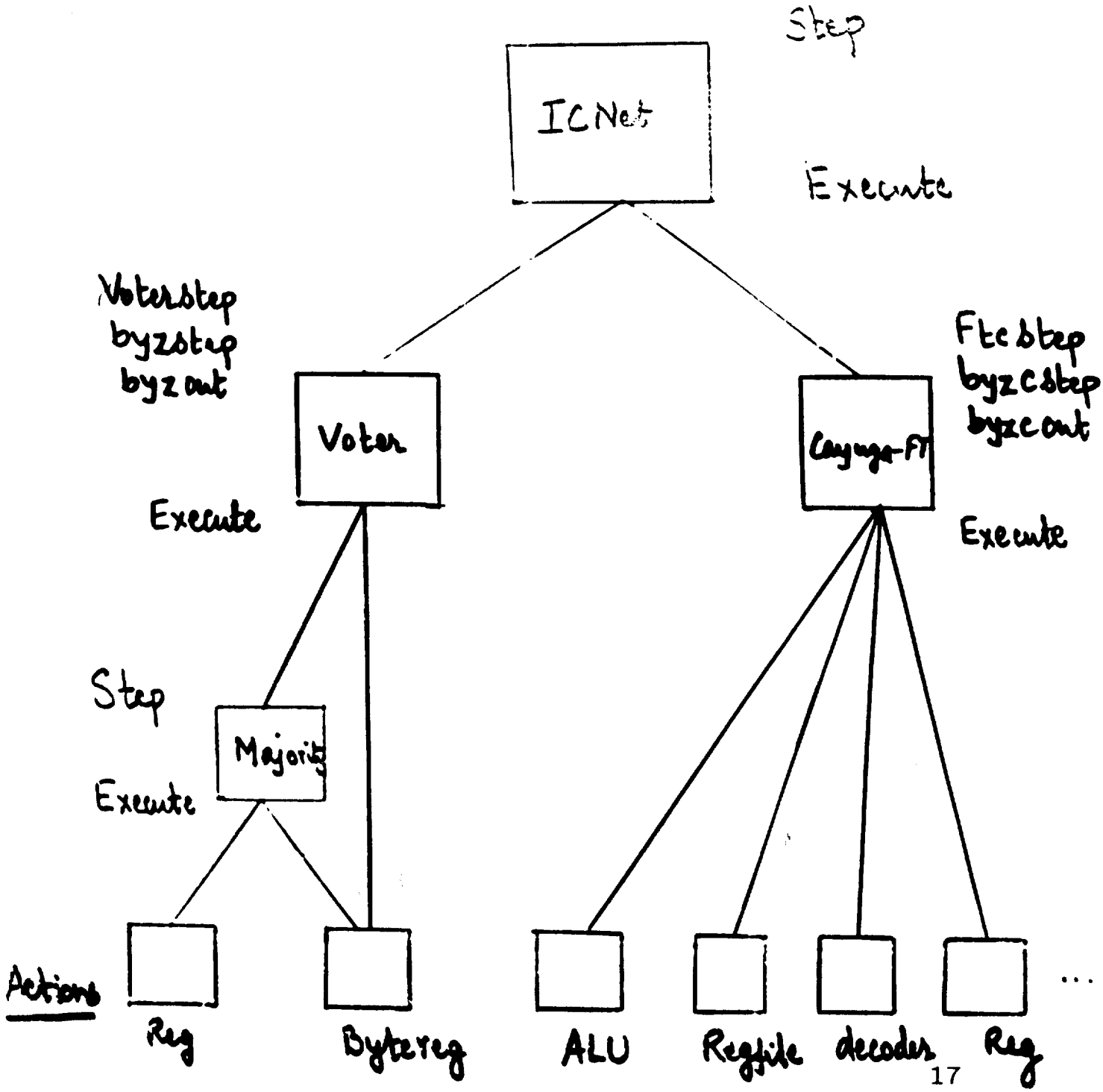


  - Advance conditions



- Proof strategy: "controlled symbolic evaluation (rewriting) with selective case-splits"

The Specification Hierarchy



Rationale for the hierarchy

- Decompose proofs into manageable units
- Need for the black level
  - introduce "error" actions
  - type of `Execute` is different from that of `action`
- Implication of intermediate levels
  - *pro*: proof can take "bigger" steps
  - *con*: must come up with intermediate abstract specification

Top Level Specification

```
||IcNetState ::~ <<(INDEX -> FTCstate),
||                 (INDEX -> Voterstate), Interrupts>>

IcNetStep <<ftc,vtr, int:rest>> =
     <<newftc,newvtr ,rest>>
     where newftc index
            = fault_ftc_step index ftc (ftcinput index)
           newvtr index
            = fault_vtr_step index vtr (vtrinput index)
           ftcinput index
            = make_ftc_in (select_int index int)
                          (fault_to_proc index ftc vtr)
           vtrinput index
             = Voterinput index ftc vtr
                                    (ftcinput index )

fault_ftc_step index s in =
    FtCayugaStep (s index) in , ~(faulty index)
    byzCayugaStep (s index) in

fault_vtr_step index s =
    voterstep (s index) , ~(faulty index)
    byzstep (s index)

...
```

Formal Statement of Correctness

```
MainTheorem :=
    Preconditions `s` => ResultConsistent `s`

ResultConsistent `s` :=
    Consistent `icvec s (Iterate #12 IcNetStep s)`

Consistent `array` :=
  `faulty index`=`False` =>
              IndexConsistent `array` `index`


IndexConsistent `array` `index` :=

    (`faulty (succ index)`=`False`=>
       `(array index).succ`=`array (succ index)`)

  & (`faulty (succ2 index)`=`False`=>
       `(array index).succ2`=`array (succ2 index)`)

  & (`faulty (succ3 index)`=`False`=>
       `(array index).succ3`=`array (succ3 index)`)
```

```
Preconditions 's' :=
    Proper_icnet 's' & Sync 'LDP1' 's' & All_go 's'


Sync 'cs' '<<ftc,vtr,inlist>>' :=
   ('faulty ONE' = 'False' =>
           'control (vtr ONE)'='cs')
 & ('faulty TWO' = 'False' =>
           'control (vtr TWO)'='cs')
 & ('faulty THREE' = 'False' =>
           'control (vtr THREE)'='cs')
 & ('faulty FOUR' = 'False' =>
           'control (vtr FOUR)'='cs')


All_go 's' :=
   ('faulty ONE'='False' =>
      ('go_of (vtr s ONE)'='False' & 'go_signal s ONE'='GO
 & ('faulty TWO'='False' =>
      ('go_of (vtr s TWO)'='False' & 'go_signal s TWO'='GO
 & ('faulty THREE'='False' =>
      ('go_of (vtr s THREE)'='False' & 'go_signal s THREE'
 & ('faulty FOUR'='False' =>
      ('go_of (vtr s FOUR)'='False' & 'go_signal s FOUR'='
```

```
Preconditions 's' :=
  Proper_icnet 's' & Sync 'LDP1' 's' & All_go 's'


Sync 'cs' '<<ftc,vtr,inlist>>' :=
   ('faulty ONE' = 'False' =>
           'control (vtr ONE)'='cs')
 & ('faulty TWO' = 'False' =>
           'control (vtr TWO)'='cs')
 & ('faulty THREE' = 'False' =>
           'control (vtr THREE)'='cs')
 & ('faulty FOUR' = 'False' =>
           'control (vtr FOUR)'='cs')


All_go 's' :=
 ('faulty ONE'='False' =>
   ('go_of (vtr s ONE)'='False' & 'go_signal s ONE'='GO'))

& ('faulty TWO'='False' =>
   ('go_of (vtr s TWO)'='False' & 'go_signal s TWO'='GO'))

& ('faulty THREE'='False' =>
   ('go_of (vtr s THREE)'='False'
                          & 'go_signal s THREE'='GO'))

& ('faulty FOUR'='False' =>
   ('go_of (vtr s FOUR)'='False'
                          & 'go_signal s FOUR'='GO'))
```

The proof strategy reused

"controlled symbolic execution of design"

1. Instantiate the states of components and inputs with appropriate symbolic constants.

2. Add all the conditions on the constants implied by the preconditions of the theorem as hypothesis.

3. Symbolically evaluate design.

4. Try *case-splitting* on all the conditionals automatically.

5. If either of the previous two steps seem to take too long, then case-spilt on the controller states and inputs before symbolic evaluation (step 3).

New technology needed

- Modeling faulty behavior
- Specification
  - determining the right hierarchy
  - writing intermediate "abstract" spec
  - defining abstraction function (ABS)
- Proof: "design level properness" implies "abstract level properness"

General Observations

- An engineering-oriented verification experience
  Lilith → MiniCayuga → IC circuit

- Methodology: top-down + bottom-up

- Level of effort: 1 man year
  - building the tool
  - developing designs
  - verification

Verification Effort Milestones

- formulated a top level correctness statement
- designed and verified a simple voter circuit
- specified voter and processor for a continuous voting scheme
- designed and verified second voter design

- discovered continuous voting scheme was "hard to synchronize"
- respecified voter and processor for a voting-on-demand scheme
- redesign and reverify voter
- verified overall system
- verified processor

- To integrate theorem proving based verification technology into the design process we need:
  - more machine assistance
  - domain specialization
- The next step ?
  - A useful way of reporting failed proof attempts
  - Interaction with motivated and patient engineering design teams and projects